Formalizing a SAT Proof Checker in Coq

Ashish Darbari, Bernd Fischer
School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, England
email: {*ad06v,b.fischer*}*@ecs.soton.ac.uk*

Joao Marques-Silva
School of Computer Science and Informatics
University College Dublin, Belfield, Dublin 4, Ireland
email: *jpms@ucd.ie*

Abstract

Advances in SAT technology have made it possible for the SAT solvers to solve much bigger instances of problems using fewer resources. Much of the speed of these solvers comes from well-crafted optimizations but these complicate the implementations of the solvers, and make them vulnerable to bugs. However, assurance can be re-gained by use of a checker that validates the outcome of the solver. Two important aspects of this approach are (i) to ensure that the checker program itself is bug free, and (ii) is easy-to-use as a standalone executable.

We have designed and implemented a SAT proof checker using the Coq proof assistant. Our checker is capable of validating a SAT or an UNSAT claim of a SAT solver. In this paper we report on the more interesting aspect of checking the unsatisfiability claims, which have the form of a ground resolution proof. We present our formalization of the checker as a set of definitions within Coq, and characterize and prove its correctness properties. The proofs have been all machine checked in Coq, and an equivalent Ocaml executable program is extracted that can be used independently of the proof-assistant itself. Finally, we present some early evaluation results on industrial benchmarks to illustrate the strength of the extracted checker.

1 Introduction

Advances in SAT technology have made it possible for SAT solvers to be routinely used in the verification of large industrial problems. Moreover, they are now also used as back-end verification engines in several safety-critical domains such as railway systems [1] and avionics [2]. Such applications require some form of formal certification or guarantee that they are correct.

However, much of the performance enhancements in SAT technology come from well-crafted optimizations that make the SAT solvers vulnerable to implementation bugs. At the same time their complexity makes formal proofs of their correctness extremely difficult. For example, Lescuyer et al [3] formalized a SAT solver in the Coq proof assistant and extracted an executable program. The resulting program was mathematically rigourously checked, but its performance suffered, due to the lack of optimizations. Reasoning about these optimizations makes the formal correctness proofs exceedingly hard, as shown by Marić [4], who verified the pseudo-code of the SAT algorithm used in the ARGO-SAT solver but did not verify the solver itself.

An alternative, and more effective approach for ensuring correctness is to not formalize the SAT solver itself, but to instead formalize an independent *checker* in a proof-assistant, and use that checker to validate the outcome of the SAT solver. Weber and Amjad [5] proposed the idea of checking resolution proofs from SAT solvers by re-constructing them in LCF style higher-order logic theorem provers Isabelle, HOL 4, and HOL Light. They imported the proof trace output obtained from the proof-logging versions of Zchaff and Minisat into these theorem provers, and re-played the proofs to check whether they are valid. The benefit of this approach is that they can rely on the trusted LCF style kernel of the theorem provers to check the resolution proof obtained from the trace. However, a problem of this approach is that users need to be able to use these theorem provers in order to use the checker.

Our approach follows the general ideas of Weber and Amjad, but solves the (practical) problem of their work by extracting a stand-alone checker that can be used independently of the proof assistant. We have formalized and implemented a SAT checker called SHRUTI in Coq. Given a CNF description of the problem, and a proof trace obtained from a SAT solver, our checker can determine the validity of the claim made by the solver. Our formalization has two parts, one for checking the satisfiability claim (SAT), and another to validate the unsatisfiability claim (UNSAT). In this paper we present the formalization of the UNSAT part of the checker in the Coq proof assistant. We present some preliminary evaluation results on the industrial benchmarks from the SAT Race competition [6] to illustrate the strength of our approach.

2 Proof Checking Overview

Most SAT solvers can also produce a proof carrying the explanation about why the given problem was unsatisfiable when they produce an UNSAT answer. Any checker should be able to read these proof traces and should come up with a Yes/No answer depending on whether an outcome of the SAT solver is correct or not. In fact many of these solvers such as Zchaff, Minisat, Picosat and Booleforce provide a checker that does just that. However, none of these checkers are formally certified for correctness.

An UNSAT proof trace is a representation of general resolution proofs consisting of the original clauses used during resolution and the intermediate resolvents obtained by resolving the original input clauses. The parts of the proof which are regular input resolutions are called chains. The whole trace thus consists of original clauses and chains. Since a chain is a new proof rule, its input clauses are called 'antecedents' and the final resolvent simply 'resolvent'.

In order to design an efficient checking algorithm we made use of the resolution inference rule [7]. This rule takes a pair of clauses in disjunctive normal form, and produces a union of the two clauses, cancelling any complementary literals present in the two clauses. Of course, it is assumed that the input clauses themselves have no duplicate literals, and have no complementary literals within themselves. It is well known that this inference rule is sound and complete for propositional logic and the proof can be found in [8, 9]. When this inference rule is used to compute a resolution derivation on a set of clauses such that each resolved variable (i.e., the variable that occurs in the pair of complementary literals) is distinct and each clause is either an input clause or a derived clause obtained by the application of the resolution rule, the resolution derivation is called trivial resolution [10]. We often use the term 'trivial resolution' to mean the application of the 'resolution inference rule' since the application of the latter results in a trivial resolution.

The use of resolution rule ensures that the number of resolution steps taken to compute the final resolvent of a chain is linear with respect to the number of antecedents within the chain. Thus the computation of a final resolvent in a chain begins at one end of the chain (in our case left most end of the chain) and uses each antecedent within the chain only once.

We decided to test our certified checker by reading the proof trace formats generated by Picosat, because it can also generate proof traces readable in ASCII form as compared to some of the other proof logging versions of solvers that only produces binary versions. Picosat [11] was also voted as one of the best SAT solvers in the industrial category of SAT Race 2007. Like many SAT solvers, Picosat reads the problem representation in DIMACS [12] notation. This uses non-zero integers to denote literals. A positive variable is denoted by a positive integer while its complement uses a negative integer. Zeroes are only used as delimiters. As an example consider the following unsatisfiable formula adapted from the `README.tracecheck` file distributed with Booleforce. It consists of all possible binary clauses over the two variables 1 and 2.

```
 1   2  0
-1   2  0
 1  -2  0
-1  -2  0
```

The zeroes at the end of rows are delimiters. A Picosat proof trace consists of such rows representing the input clauses, followed by rows encoding the proof chains. Each "chain row" consists of an asterisk (*) as place-holder for the chain's resolvent,[1] followed by the identifiers of the clauses involved in the chain. Each chain row thus contains at least two clause identifiers, and denotes an application of one or more of the resolution inference rule, describing a trivial resolution derivation. Each row also starts with a non-zero positive integer denoting the identifier for that row's (input or resolvent) clause. In an actual trace there are additional zeroes as delimiters at the end of each row, but we remove these before we start proof checking. The input to our checker thus looks as follows:

```
1   1   2
2  -1   2
3   1  -2
4  -1  -2
5   *   3  1
6   *   4  2  5
```

The first four rows denote the input clauses from the original problem (see above) that are used in the resolution, with their identifiers referring to the original clause numbering, whereas rows 5 and 6 represent the proof chains. In row 5, the clauses with identifiers 3 and 1 are resolved using a single resolution rule, whilst in row 6 first the original clauses with identifier 4 and 2 are resolved and then the resulting clause is resolved against the clause denoted by identifier 5 (i.e., the resolvent from the previous chain), in total using two resolution steps.

The algorithm of checking the validity of the proof trace relies singularly on the repeated use of the resolution rule. Checking begins at the first row of the proof chain (in the above example it would be 5), and the resolution rule is applied to all the antecedent clauses denoted by the identifiers in the chain. The resolvent clause (in this case consisting of $\{1\}$) is stored in a lookup table and is tagged with the key identifier 5. This process is then repeated for the next identifier in the proof chain (in our example it would be 6) and after two resolution rule applications an empty clause is obtained. If the empty clause is obtained then the given problem is UNSAT (i.e., the checker will

[1]This is generated by Picosat; there is another option of generating proof traces from Picosat where instead of the asterisk the actual resolvents are generated delimited by a single zero from the rest of the chain.

return a Yes answer), or else if all proof chain identifers have been checked and the empty clause is not derived, the given problem is not UNSAT (i.e., the checker will return a No answer). Correctness of the checking algorithm depends on the correct implementation of the resolution inference rule. The resolution rule itself is correct if it satisfies the following conditions:

1. All complementary literals are deleted from the given pair of clauses.
2. If the input pair of clauses contains a common literal then only one copy of that literal is made in the resolvent.
3. All unequal literals in the given pair of clauses are retained in the resolvent.

Additionally the resolution rule should produce an empty resolvent for a given pair of clauses that only contain complementary literals. A correctly implemented proof-checking algorithm would produce an empty clause from a proof trace for an unsatisfiable problem provided the trace contains a well-ordered chain of antecedents. If the ordering of the antecedents is not preserved which is the case with the compact resolution trace produced by Picosat, we are likely to introduce a scenario at the time of checking in which each trivial resolution step does not necessarily create a resolvent by cancelling complementary literals using linear number of steps. In other words the antecedents cannot be resolved to form a regular input resolution proof or the trivial proof efficiently. The *tracecheck* program distributed with Picosat can expand the trace output from Picosat and fix the ordering problem of the chains. It takes a resolution proof trace from Picosat as input and creates an extended resolution trace.

In the next section we present our formulation of the UNSAT part of the checker SHRUTI specifically showing the formalization of the resolution inference rule and we characterize its correctness properties by formalizing three main theorems.

3 Formalization of SHRUTI in Coq

3.1 Motivation for using Coq

We wanted to design a certified proof checker that can be formalized and mechanically verified using a proof-assistant to generate a high level of confidence, and at the same time enable the user to use it independently of the proof-assistant. We envision that by not compromising the safety, and enhancing ease-of-use, we can encourage the use of certified checkers as a regular component during the SAT checking flow. We therefore decided to use a proof-assistant in which it would be possible to achieve both our goals and the obvious choice was the Coq proof assistant. Coq has been widely used in several certification projects; most well known is the certification of a C compiler [13].

3.2 Formalizing SHRUTI

At the heart of SHRUTI is the formalization of the UNSAT part of the checker in Coq. The formalization makes use of a shallow embedding of the proof checker inside Coq using the data types and data structures of the Coq meta logic to represent the types and data structures of the proof checker. We then formulate definitions over these, and formally prove inside Coq that these definitions are correct. Once the Coq formalization is complete, Ocaml code is extracted from it through the extraction API that comes with Coq. At the time of extraction, the Coq data types/data structures are mapped to Ocaml data types/data structures. This way, we get the safe, static, one-off characterization in Coq combined with the run-time execution speed of Ocaml. The extracted Ocaml code expects to read its input data from data structures such as tables and lists. Data is stored in these from files containing the CNF description and the proof trace. This is handled by some extra piece of Ocaml glue that wraps the extracted Ocaml code. The glue code also contains functions for profiling and logging the results in files. The result is then compiled into a native machine code executable that can be run independently of the proof-assistant Coq. A high-level architectural view is shown in Figure 1.

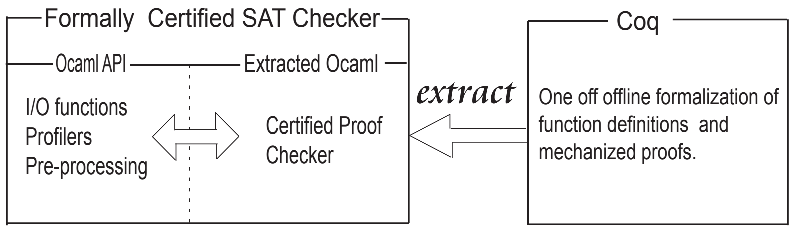
Formally Certified SAT Checker
Ocaml API
Extracted Ocaml
I/O functions
Profilers
Pre-processing
Certified Proof Checker
extract
Coq
One off offline formalization of function definitions and mechanized proofs.

Figure 1: A High Level Architectural View of the Certified Checker.

We assume familiarity with the quantifiers (∀, ∃) and logical connectives such as *and* (∧), *or* (∨), *not* (¬) and implication. We distinguish implication over propositions by ⊃ and over types with → for presentation clarity, though inside Coq they are exactly the same. The notation ⇒ is used during pattern matching (using match-with-end) as in other functional languages. For type annotation we use :, and for the cons operation on lists we use ::. Empty list is denoted by *nil*. The set of integers is denoted by Z, the type of polymorphic list by *list* and the type of list of integers by *list* Z. List containment is represented by ∈ and its negation by ∉. The function *Zabs* computes the absolute value of an integer. We use Definition to denote the Coq function definitions. Main data structures that we have used in Coq formalization are lists, and finite maps. Finite maps or simply maps are functionally similar to hashtables with integer keys and polymorphic bindings although they are implemented using balanced binary trees.

To define the resolution function we make use of an auxillary function *union* which is defined below. This function takes as input a pair of clauses represented as a list of integers and an accumulator, and performs the functionality of the resolution operation.

```
Definition union (c1 c2 : list Z)(acc : list Z) =
match c1, c2 with
    | nil, c2 ⇒ app (rev acc) c2
    | c1, nil ⇒ app (rev acc) c1
    | x :: xs, y :: ys ⇒ if (x + y = 0) then union xs ys acc
        else if (Zabs x) < (Zabs y) then union xs (y :: ys) (x :: acc)
        else if (Zabs y) < (Zabs x) then union (x :: xs) ys (y :: acc)
        else union xs ys (x :: acc)
```

The key feature of this function is that it expects the input clauses to be sorted by absolute value and the resolvent produced is also sorted by absolute value. This has the benefit of keeping the efficiency of the resolution operation linear in the size of the input clauses.

```
Definition sorted =
Inductive sorted : list Z → Prop :=
| sorted0 : sorted nil
| sorted1 : ∀z : Z. sorted (z :: nil)
| sorted2 : ∀z1 z2 :Z ℓ: list Z. (Zabs z1 ≤ Zabs z2) ⊃ sorted (z2 :: ℓ) ⊃ sorted (z1 :: z2 ::ℓ)
```

Note that in this predicate we do not enforce the constraint that an element has to be strictly less than the other, as we use the $\leq$ relation. However, when it comes to the proofs later on, this constraint is automatically enforced by stating that the clauses cannot contain duplicates or complementary literals.

Once the input clauses are sorted by absolute value, at the time of resolution each integer in the clause is compared pointwise. If the integers are complementary to one another then neither is added to the accumulator else the smaller (in terms of absolute value) of the two is added. If the integers are equal, one of them is stored in the accumulator. Once a single run of any of the clauses is finished, the accumulator's contents are sorted and then merged with the other, longer clause. Sorting is done by simply reversing the accumulator. This is because integers are added to the front of the list using the (::) operation, and the resulting accumulator has the final elements in descending order.

The actual binary resolution function is defined below. It is denoted by $\bowtie$ and makes use of the *union* function.

Definition $c_1 \bowtie c_2 = (\mathit{union}\ c_1\ c_2\ \mathit{nil})$

To ensure that the formalization of our checker is correct we need to check that the resolution ($\bowtie$) function is defined correctly. What it means is that it should preserve the basic properties of the binary resolution function which are enumerated below:

1. Any pair of complementary literals is deleted in the resolvent obtained from resolving a given pair of clauses (Theorem 1).
2. All non-complementary literals that are pairwise unequal are retained in the resolvent (Theorem 2).
3. For a given pair of clauses, if there are no duplicate literals within each clause, then for a literal that exists in both the clauses of the pair, only one copy of the literal is retained in the resolvent (Theorem 3).

We have proven these properties in Coq. The actual proof, including several lemmas, comprises in total about 2000 lines of proof script in Coq.

For the sake of clarity in presentation we do not detail all the assumptions but we need to assume that the following assumptions hold for the three main theorems that we present later on.

1. No duplicates are allowed in each of the clauses c_1 and c_2.
2. There exists no mutually complementary pair of literals within each of the clauses c_1 and c_2.
3. No Zeros are allowed in the clauses c_1 and c_2.

These assumptions are essentially the constraints imposed on input clauses when the resolution function is applied in practice.

Since we have developed the machine checked proofs[2] we will not show the proof of these theorems in this paper. The general stratgey is to use structural induction on clauses c_1 and c_2. For each theorem, this results in four main goals, three of which are proven by contradiction since for all elements ℓ_1, $\ell_1 \notin \mathit{nil}$. For the remaining goal a case-split is done on if-then-else, thereby producing 8 sub-goals, some of whom are proven from induction hypotheses, and some from conflicting assumptions arising from the case-split. For others we employ a collection of special properties. Some of these are about integers and their reationship with their absolute values

[2]Available at http://users.ecs.soton.ac.uk/ad06v/papers/coqwkshp09/

and the fact that these values appear in sorted lists without duplicates. Some are about counting an element and its relationship with the $\bowtie$-function. The interested reader is referred to the online proof script. We will point out some of the main properties used in the proof of the theorems when we present the theorem.

Theorem 1. *All complementary literals are deleted:*

$$\begin{aligned}&\forall c_1\ c_2.\ sorted\ c_1 \supset sorted\ c_2 \supset\\&\quad \forall \ell_1\ \ell_2.\ (\ell_1 \in c_1) \supset (\ell_2 \in c_2) \supset\\&\qquad (\ell_1 \notin c_2) \supset (\ell_2 \notin c_1) \supset (\ell_1 + \ell_2 = 0) \supset\\&\qquad\quad (\ell_1 \notin (c_1 \bowtie c_2)) \wedge (\ell_2 \notin (c_1 \bowtie c_2))\end{aligned}$$

We make use of two important properties to prove two of the sub-goals arising in the proof of this theorem. The first property states that if an element is not present in either of c_1, c_2 or acc then it cannot be present in the resolvent of c_1 and c_2. The other important property states that if an element is already in acc then it exists in the resolvent of c_1 and c_2.

For the following theorem we need to assert in the assumption that for any literal in one clause there exists no complementary literal in the other clause. This is defined by the predicate *NoMutualComp*.

Theorem 2. *All non-complementary, unequal literals are retained:*

$$\begin{aligned}&\forall c_1\ c_2.\ sorted\ c_1 \supset sorted\ c_2 \supset\\&\quad \forall \ell_1\ \ell_2.\ (\ell_1 \in c_1) \supset (\ell_2 \in c_2) \supset (\ell_1 \notin c_2) \supset (\ell_2 \notin c_1) \supset\\&\qquad (\ell_1 \neq \ell_2) \supset (NoMutualComp\ \ell_1\ c_2) \supset (NoMutualComp\ \ell_2\ c_1) \supset\\&\qquad\quad (\ell_1 \in (c_1 \bowtie c_2)) \wedge (\ell_2 \in (c_1 \bowtie c_2))\end{aligned}$$

For the proof we make use of an important property that states if an element is in clause c_1 and is not in clause c_2 and provided that its complement also does not exist in either c_1 or c_2 we will get that element in the resolvent of c_1 and c_2.

Theorem 3. *(Factoring) Only one copy of equal literal is retained:*

$$\begin{aligned}&\forall c_1\ c_2.\ sorted\ c_1 \supset sorted\ c_2 \supset\\&\quad \forall \ell_1\ \ell_2.\ (\ell_1 \in c_1) \supset (\ell_2 \in c_2) \supset\\&\qquad (\ell_1 = \ell_2) \supset ((\ell_1 \in (c_1 \bowtie c_2)) \wedge (count\ \ell_1\ (c_1 \bowtie c_2) = 1))\end{aligned}$$

The proof of this theorem makes use of an important counting property. It states that if an element occurs in the accumulator acc once, and it exists in $union\, c_1\, c_2\, acc$ but it does not exist in c_1 or c_2, then it must only occur once in $union\, c_1\, c_2\, acc$.

In order to check the resolution steps for each row, we have to collect the actual clauses corresponding to their identifiers and this is done by the *findClause* function.

```
Definition findClause acc ctbl rtbl dlst =
    match dlst with
     | nil ⇒ (List.rev acc,true)
     | (x :: xs) ⇒
     match (find x rtbl) with
     | Some a ⇒
        findClause (a :: acc) ctbl rtbl xs
     | None ⇒
```

```
        match (find id ctbl) with
            | None ⇒ (acc,false)
            | Some a ⇒ findClause (a :: acc) ctbl rtbl xs
```

The function *findClause* takes a list of clause identifiers (*dlst*), an accumulator (*acc*) to collect the list of clauses, and requires as input a table that has the information about all the input clauses (*ctbl*). It also takes another table (*rtbl*) as an argument which is the table that contains the processed resolvents. Whenever a clause id is processed, then its resolvent clause is first looked up in the resolvent table, if that contains no entry for the given clause identifier, the clause is obtained from *ctbl*. If there is no entry in either of the tables, an error is signalled. It means there is a clause id for which there is no clause. This could be because there is an input/output problem with the proof trace file.

We then prove some sanity-checking properties about the maps. An obvious property that follows from the finite map implementation itself is that if a key is inserted in a table it will return some binding on being queried. We prove that if an entry is not found in the clause table and the resolvent table then the false flag is raised.

The function that uses the ⋈ function recursively on a list of input clause chain is called *hyperResolution* and it simply folds the ⋈ function from left to right for every row in the proof part of the proof trace file.

```
Definition hyperResolution lst =
  match (lst : list (list Z)) with
    | nil ⇒ nil
    | (x :: xs) ⇒ List.fold_left (⋈) xs x
```

The function *findAndResolve* is our last function defined in Coq world for UNSAT checking and provides a wrapper on other functions. The proof traces obtained from Picosat contains the proof chains specifying the clause identifiers used to derive the conflict, and the actual clauses that are used to generate the conflict. At the time of proof checking the pre-processed (trailing 0s removed) input proof trace is scanned and for each line in the trace it is either stored into a clause table (*ctbl*) since it represents an input clause, or in the trace table (*ttbl*) because it denotes a proof chain. The function *findAndResolve* then starts the checking process by first snarfing all the antecedents (identifiers for clauses) in a chain from the trace table, and then for each antecedent, obtains the actual clause either from the clause table or from the resolvent table by using the function *findClause*.

```
Definition findAndResolve ctbl ttbl rtbl id =
  let dlst = (find id ttbl) in
    match dlst with
      | None ⇒ (add id (0 :: nil) rtbl)
      | Some a ⇒
        let (cls,flag) =
        findClause nil ctbl rtbl a in
        match flag with
        | false ⇒ add id (0::(0::nil)) rtbl
        | true ⇒ add id (hyperResolution cls) rtbl
```

Once all the clauses are obtained for a single chain, the function *hyperResolution* is called and applied on the list of clauses for all proof chains. For each chain, the resolvent is stored in a separate resolvent table and tagged with the chain identifier from the trace table. It then checks whether the resolvent for the identifier of the last chain is an empty clause (i.e., empty list), and returns Yes (meaning that the solver's UNSAT claim is valid) if it finds one, else No.

We prove that if there is no binding for a given identifier in the trace table then a list with single zero is inserted in the resolvent table corresponding to this identifier. Similarly we prove that if the *findClause* function returns an error (flag is set to false) then a list with two zeroes is inserted in the resolvent table.

Since the proof trace obtained from Picosat contains proof chains that are a trivial resolution derivation, and since they are well-ordered, it is guaranteed that at the time of proof checking each application of the resolution inference rule will resolve at least one complementary pair of literals, thereby decreasing the count of total literals in the resolvent. For an UNSAT problem there would be enough proof chains and enough antecedents in each chain so that finite amount of resolution inference rule applications would eventually produce a pair of clauses with equal number of literals that are complementary to one another, and thus the final application of resolution rule would produce an empty clause. Our implementation of the resolution inference rule guarantees (due to the three main theorems presented) that this will happen provided the input proof trace is well-ordered and represents a chain of trivial resolution derivations.

We also check whether the given proof trace is a legal proof trace, i.e., whether any input clause used in any proof chain in the given trace is contained in the original problem. If the trace is not legal then the user gets a message and the checker aborts. However, this is provided as an option to the user at runtime, and if invoked, adds about 1-2% time overhead. This feature is currently optional because the uncertified checkers currently do not do this, and for comparing the runtime performances of our checker with uncertified ones we would like to disable this option at runtime to keep the comparison fair. Comparison results for these are still in process.

4 Results and Discussion

Our checker was tested on a chosen set of industrial benchmarks from SAT Race. The results are summarized in Table 1. We pre-processed the input trace file to remove trailing zeroes using Ocaml routines. We do not enforce the check for duplicates in the input trace. If they are present in any line of the trace, they will "ripple out" in the resolvent, and an empty clause cannot be derived using the resolution based proof-checking. The input trace then no longer represents an UNSAT problem and the checker will simply return a No verdict.

We experimented with the extraction process and optimized the extracted Ocaml functions for efficiency. In our first implementation we only mapped the Coq lists to Ocaml lists. The resulting implementation (shown as SHRUTI (orig.) in the table) was more than one order of magnitude slower.

The Coq Zs were replaced with Ocaml integers and we replaced the Coq functions on Zs with the equivalent Ocaml functions. We also replaced the Coq finite maps with Ocaml finite maps and together with this change noticed a significant improvement.

Replacing Coq Zs with Ocaml integers and the maps gave a performace boost by a factor of 5-10. This can be perhaps attributed to the reduced overhead when dealing with Ocaml integer keys (in the Ocaml maps) directly, without having to convert between Coq Zs and Ocaml integers, and that all integer operations were now done on Ocaml integers.

A substantial bottle-neck in performance was the Ocaml garbage collector that unwittingly kicked in each time the number of inference steps exceed one million. The effect of this was almost an

Benchmark	Proof Steps	SHRUTI (opt.)	SHRUTI (orig.)
een-tip-uns-nusmv-t5.B	122816	0.91	5.82
een-pico-prop01-75	246430	1.29	10.14
ibm-2004-26-k25	1132	0.004	0.02
ibm-2002-26r-k45	1105	0.001	0.02
ibm-2004-3_02_1-k95	114794	0.63	3.87
ibm-2004-6_02_3-k100	126873	0.76	4.92
ibm-2004-1_11-k25	254544	1.86	11.29
ibm-2002-07r-k100	255159	1.38	9.37
ibm-2004-2_14-k45	701430	6.59	51.51
manol-pipe-c10nidw_s	458042	2.82	19.51
manol-pipe-f6bi	1058871	10.32	97.85

Table 1: Results showing the times taken by our extracted checker on a sample of industrial benchmark problems from the SAT Race Competition. We show the number of proof steps obtained from Picosat/tracecheck in the second column. The third column shows the time taken by our optimized version SHRUTI (opt.) whilst the last one shows the timings obtained from the originally extracted, un-optimized version SHRUTI (orig.). The compiled binaries in both the cases were executed on a server running Red Hat Linux with Intel Xeon CPU 3 GHz, and 4GB memory. All times include resolution checking time, I/O and pre-processing times.

exponential drop in performance. We therefore changed the runtime settings of the garbage collector, by specifying large initial sizes for major and minor heaps and controlling the `space_overhead` and `max_overhead` settings such that minimal amount of garbage collection takes place. This enabled us to have much better execution times that scaled linearly with the number of inferences and we are now able to check proof traces with up to 15 million inferences. The results for this are shown as SHRUTI (opt.) in the table. Our extraction process only mapped Coq data structures to Ocaml data structures to enhance efficiency which is a standard practice in any program extraction based development in Coq. An important point to note here is that the core logic and functionality of the checker program is not compromised by program extraction in Coq.

5 Related Work

Lescuyer et al. [3] formalized a SAT solver in the Coq proof assistant and extracted an executable program. The resulting program was mathematically rigourously checked, but its performance suffered, due to the lack of optimizations. Recently Marić [4] proposed to verify a SAT solver with the low-level optimizations. He formalized the ARGO-SAT solver in Isabelle/HOL by modelling the solver and its low-level optimizations at an abstract level (pseduo code). One major difficulty is that it is difficult to formalize low-level optimizations (that work on real code used in a SAT solver) at a sufficiently abstract level without loosing sight of the low-level details. Even though optimizations were formalized, they were done for the pseudo-code, not the actual code that is used in the solver which still leaves the gap between what is formalized and what is used at runtime. Moreover, its practically not very useful (it took Marić one man year) to verify a solver, since it has to be done for each new solver. Instead its more efficient to verify a checker correct (since checkers are small and relatively straight-forward), and use it to validate outputs of any solver that produces proof certificates that were previously agreed.

Weber and Amjad [5] proposed the idea of checking resolution proofs from SAT solvers by re-constructing them in higher-order logic based theorem provers Isabelle, HOL 4 and HOL Light. They imported the proof trace output obtained from the proof-logging versions of Zchaff and Minisat into these theorem provers, and re-played the proofs to check if they are valid.

A key difference between our checker and Weber and Amjad's is in the design and usage. In order to use Weber and Amjad's checker one has to have the different theorem provers installed, and more importantly the knowledge of using each one of them becomes paramount. In our case, we provide an executable binary that can be run independently of the Coq theorem prover or any other for that matter. Thus usability is considerably enhanced in our case. Weber and Amjad mostly reported their performance results on pigeon-hole problems and not much on industrial benchmarks. Pigeon hole problems though somewhat hard are also artificially created and thus share a common structure to them, so we personally decided not to calibrate our checker on these problems and instead chose to test ours on industrial benchmarks from the SAT Race Competition. We are investigating if we can get Weber and Amjad's checkers results' on the industrial benchmarks, and provide some comparison with our's. Bulwahn et al. [14] experimented with the idea of doing reflective theorem proving in Isabelle and suggested that it can be used for designing a SAT checker. In this sense their work is closest to our's. They proposed to enhance the functional core of Isabelle with imperative data structures for efficiency. However we have not seen the complete formalization of their SAT checker, and no benchmark results have been reported to the best of our knowledge. Recently there has been some work done in certifying SMT solvers notable amongst them are the work done by Moskal [15] and de Moura [16].

In a recent development related to Coq, there has been an emergence of a tool called Ynot [17] that can deal with arrays, pointers and file related I/O in a Hoare Type Theory. Future work in certification using Coq should definitely investigate the usage of this.

6 Conclusion

We presented the formalization of a SAT checker in the Coq proof assistant and presented some of the early benchmarking results. We observed that by using Coq we could do a one-off offline formalization of the checker and machine check all the proofs in Coq, while at the same when we extract an ocaml program, we obtain a fast executable binary, that can be used for checking industrial benchmarks as demonstrated by some of our results.

Acknowledgement

We thank Yves Bertot, Pierre Letouzey, and many more people on the Coq mailing list who helped us with Coq questions. We especially thank Tjark Weber and Hasan Amjad for answering our questions on their work. This work was funded by EPSRC, Grant # EP/E012973/1.

References

[1] M. Penicka, "Formal Approach to Railway Applications," in *Formal Methods and Hybrid Real-Time Systems*, Lecture Notes in Computer Science 4700, pp. 504–520. Springer, 2007.

[2] J. Hammarberg and S. Nadjm-Tehrani, "Formal Verification of Fault Tolerance in Safety-Critical Reconfigurable Modules," *International Journal on Software Tools for Technology Transfer* 7(3), pp. 268–279, 2005.

[3] S. Lescuyer and S. Conchon, "A Reflexive Formalization of a SAT Solver in Coq," in *Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics*, 2008.

[4] F. Marić, “Formalization and Implementation of Modern SAT Solvers,” *Journal of Automated Reasoning* 43(1), pp. 81–119, 2009.

[5] T. Weber and H. Amjad, “Efficiently Checking Propositional Refutations in HOL Theorem Provers,” *Journal of Applied Logic* 7(1), pp. 26–40, 2009.

[6] “SAT Race Competition,” 2008. [Online]. Available: http://baldur.iti.uka.de/sat-race-2008/index.html

[7] J. A. Robinson, “A Machine-Oriented Logic Based on the Resolution Principle,” *Journal of ACM* 12(1), pp. 23–41, 1965.

[8] J. A.Robinson, *Logic: Form and function - The Mechanization of Deductive Reasoning.* Elsevier, 1980.

[9] C.-L. Chang and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1997.

[10] P. Beame, H. Kautz, and A. Sabharwal, “Towards Understanding and Harnessing the Potential of Clause Learning,” *Journal of Artificial Intelligence Research* 22, pp. 319–351, 2004.

[11] A. Biere, “PicoSAT Essentials,” *Journal on Satisfiability, Boolean Modeling and Computation* 4, pp. 75–97, 2008.

[12] “Satisfiability Suggested Format,” 1993. [Online]. Available: ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/

[13] X. Leroy and S. Blazy, “Formal Verification of a C-like Memory Model and its uses for Verifying Program Transformations,” *Journal of Automated Reasoning* 41(1), pp. 1–31, 2008.

[14] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews, “Imperative Functional Programming with Isabelle/HOL,” in *Proc 21st International Conference on Theorem Proving in Higher Order Logics.* Lecture Notes in Computer Science 5170, pp. 134–149. Springer, 2008.

[15] M. Moskal, “Rocket-Fast Proof Checking for SMT Solvers,” in *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 4963, pp. 486–500. Springer, 2008.

[16] L. M. de Moura and N. Bjørner, “Proofs and Refutations, and Z3,” in *Proc. 7th International Workshop on the Implementation of Logics*, CEUR Workshop Proceedings 418, 2008.

[17] A. Nanevski, G. Morrisett, A. Shinnar, and P. Govereau, “Ynot: Reasoning with the Awkward Squad,” *In Proc. 13th International Conference on Functional Programming*, pp. 229–240. ACM Press, 2008.